The Embedded I/O Company

TEWS TECHNOLOGIES

TDRV015-SW-82

Linux Device Driver

Reconfigurable FPGA

Version 2.0.x

User Manual

Issue 2.0.1

August 2015

TEWS TECHNOLOGIES GmbH

Am Bahnhof 7 25469 Halstenbek, Germany

Phone: +49 (0) 4101 4058 0 Fax: +49 (0) 4101 4058 19

e-mail: info@tews.com www.tews.com

TDRV015-SW-82

Linux Device Driver

Reconfigurable FPGA

Supported Modules:

TAMC631 (TPLD001)
TAMC640 (TPLD002)
TAMC641 (TPLD003)
TAMC651 (TPLD004)
TPMC632 (TPLD005)

This document contains information, which is proprietary to TEWS TECHNOLOGIES GmbH. Any reproduction without written permission is forbidden.

TEWS TECHNOLOGIES GmbH has made any effort to ensure that this manual is accurate and complete. However TEWS TECHNOLOGIES GmbH reserves the right to change the product described in this document at any time without notice.

TEWS TECHNOLOGIES GmbH is not liable for any damage arising out of the application or use of the device described herein.

©2011-2015 by TEWS TECHNOLOGIES GmbH

Issue	Description	Date
1.0.0	First Issue	March 14, 2011
1.0.1	Supported Modules added	September 30, 2011
2.0.0	New API implemented	March 7, 2012
2.0.1	Include statement in Example Codes corrected Reference to Engineering Documentation removed	August 18, 2015

Table of Contents

1 INTRODUCTION 4

2 INSTALLATION 5

2.1 Build and install the Device Driver 5
2.2 Uninstall the Device Driver 6
2.3 Install Device Driver into the running Kernel 6
2.4 Remove Device Driver from the running Kernel 6
2.5 Change Major Device Number 7

3 API DOCUMENTATION 8

3.1 General Functions 8
3.1.1 tdrv015Open 8
3.1.2 tdrv015Close 10
3.1.3 tdrv015GetPciInfo 12
3.2 Device Access Functions 14
3.2.1 tdrv015Read8 14
3.2.2 tdrv015ReadBE16 17
3.2.3 tdrv015ReadLE16 20
3.2.4 tdrv015ReadBE32 23
3.2.5 tdrv015ReadLE32 26
3.2.6 tdrv015Write8 29
3.2.7 tdrv015WriteBE16 32
3.2.8 tdrv015WriteLE16 35
3.2.9 tdrv015WriteBE32 38
3.2.10 tdrv015WriteLE32 41
3.3 Resource Mapping Functions 44
3.3.1 tdrv015PciResourceMap 44
3.3.2 tdrv015PciResourceUnmap 46
3.4 DMA Functions 48
3.4.1 tdrv015DmaFpgaToHost 48
3.4.2 tdrv015DmaHostToFpga 51
3.4.3 tdrv015DmaGetStatus 54
3.4.4 tdrv015DmaStop 56
3.4.5 tdrv015GetDmaBuf 58
3.4.6 tdrv015FreeDmaBuf 60
3.5 Interrupt Functions 62
3.5.1 tdrv015InterruptWait 62
3.5.2 tdrv015InterruptRegisterCallbackThread 65
3.5.3 tdrv015InterruptUnregisterCallback 69
3.6 Endian Conversion Functions 71
3.6.1 endian_be16 71
3.6.2 endian_le16 72
3.6.3 endian_be32 73
3.6.4 endian_le32 74

4 DIAGNOSTIC 75

1 Introduction

The TDRV015-SW-82 Linux device driver allows the operation of the TDRV015 compatible devices conforming to the Linux I/O system specification. This includes a device-independent basic I/O interface with *open()*, *close(),*and *ioctl()* functions. The TDRV015-SW-82 device driver was designed to demonstrate the usage of main functions of the supported FPGA platform example application (e.g. TPLD001).

The TDRV015-SW-82 device driver requires Message Signaled Interrupt (MSI) support. Please make sure that your specific Linux kernel and system environment properly support MSI.

The TDRV015-SW-82 Linux device driver was designed to demonstrate the usage of main functions of the supported FPGA platform example application (e.g. TPLD001). The well documented device driver software can be used as base for customized FPGA platform applications.

The TDRV015-SW-82 device driver supports the following features:

- Read/write access to FPGA registers (8,16,32-bit big-endian)
- DMA transfer from PCI bus to FPGA PLB bus and vice versa
- Resource allocation for supported modules
- Wait for interrupts
- Register Callback functions for interrupt handling
- Driver functions are thread-safe as long as unique handles are used.

The TDRV015-SW-82 supports the modules listed below:

TAMC631	Spartan-6 AMC with FMC Module Slot (with TPLD001 FPGA Platform Example)	AMC
TAMC640	Virtex-5 AMC with FMC Slot (with TPLD002 FPGA Platform Example)	AMC
TAMC641	High Performance Virtex-5 AMC with FMC Slot (with TPLD003 FPGA Platform Example)	AMC
TAMC651	Spartan-6 FPGA AMC for MTCA.4 Rear-I/O (with TPLD004 FPGA Platform Example)	AMC
TPMC632	Reconfigurable FPGA (with TPLD005 FPGA Platform Example)	PMC

In this document all supported modules and devices will be called TDRV015. Specials for a certain device will be advised.

To get more information about the features and use of supported devices it is recommended to read the manuals listed below.

Specific Hardware User manual
Related FPGA Development Kit (FDK) documentation

2 Installation

The directory TDRV015-SW-82 on the distribution media contains the following files:

TDRV015-SW-82-2.0.1.pdf	This manual in PDF format
TDRV015-SW-82-SRC.tar.gz	GZIP compressed archive with driver source code
ChangeLog.txt	Release history
Release.txt	Information about the Device Driver Release

The GZIP compressed archive TDRV015-SW-82-SRC.tar.gz contains the following files and directories:

Directory path 'tdrv015':

tdrv015.c	Driver source code
tdrv015def.h	Driver include file
tdrv015.h	Driver include file for application program
Makefile	Device driver make file
makenode	Script for device node creation
api/tdrv015api.h	API include file
api/tdrv015api.c	API source file
example/tdrv015exa.c	Example application
example/Makefile	Example application makefile
include/config.h	Driver independent library header file
include/tpmodule.c	Driver independent library
include/tpmodule.h	Driver independent library header file
include/tpxxxhwdep.h	HAL library header file
include/tpxxxhwdep.c	HAL library source file

In order to perform an installation, extract all files of the archive TDRV015-SW-82-SRC.tar.gz to the desired target directory. The command 'tar -xzvf TDRV015-SW-82-SRC.tar.gz' will extract the files into the local directory.

- Login as *root* and change to the target directory
- Copy tdrv015.h to */usr/include*

2.1 Build and install the Device Driver

- Login as *root*
- Change to the target directory
- To create and install the driver in the module directory */lib/modules/<version>/misc* enter:

  # make install

- To update the device driver's module dependencies, enter:

  # **depmod -aq**

2.2 Uninstall the Device Driver

- Login as *root*
- Change to the target directory
- To remove the driver from the module directory */lib/modules/<version>/misc* enter:

  # make uninstall

2.3 Install Device Driver into the running Kernel

- To load the device driver into the running kernel, login as root and execute the following commands:

  # modprobe tdrv015drv

- After the first build or if you are using dynamic major device allocation it is necessary to create new device nodes on the file system. Please execute the script file *makenode* to do this. If your kernel has enabled a device file system (devfs or sysfs with udev) then you have to skip running the *makenode* script. Instead of creating device nodes from the script the driver itself takes creating and destroying of device nodes in its responsibility.

  # sh makenode

On success the device driver will create a minor device for each TDRV015 device found. The first TDRV015 device can be accessed with device node /dev/tdrv015_0, the second module with device node /dev/tdrv015_1 and so on.

The assignment of device nodes to physical TDRV015 modules depends on the search order of the PCI bus driver.

2.4 Remove Device Driver from the running Kernel

- To remove the device driver from the running kernel login as root and execute the following command:

  # modprobe –r tdrv015drv

If your kernel has enabled devfs or sysfs (udev), all /dev/tdrv015_x nodes will be automatically removed from your file system after this.

Be sure that the driver isn't opened by any application program. If opened you will get the response "*tdrv015drv: Device or resource busy*" and the driver will still remain in the system until you close all opened files and execute *modprobe –r* again.

2.5 Change Major Device Number

This paragraph is only for Linux kernels without dynamic device file system installed. The TDRV015 driver uses dynamic allocation of major device numbers per default. If this isn't suitable for the application it is possible to define a major number for the driver.

To change the major number, edit the file tdrv015def.h, change the following symbol to appropriate value, and enter `make install` to create a new driver.

TDRV015_MAJOR	Valid numbers are in range between 0 and 255. A value of 0 means dynamic number allocation.

Example:

```
#define TDRV015_MAJOR   122
```

Be sure that the desired major number isn't used by other drivers. Please check */proc/devices* to see which numbers are free.

3 API Documentation

3.1 General Functions

3.1.1 tdrv015Open

NAME

tdrv015Open – open a device.

SYNOPSIS

```
TDRV015_HANDLE tdrv015Open
(
      char          *DeviceName
)
```

DESCRIPTION

Before I/O can be performed to a device, a device handle must be opened by a call to this function. If the legacy TDRV015 driver is used, this function will also install the legacy driver and create devices with the first call. The VxBus TDRV015 driver will be installed automatically by the VxBus system.

The tdrv015Open function can be called multiple times (e.g. in different tasks)

PARAMETERS

DeviceName

This parameter points to a null-terminated string that specifies the name of the device. The first TDRV015 device is named “/dev/tdrv015_0” the second device is named “/dev/tdrv015_1” and so on.

EXAMPLE

```
#include “tdrv015api.h”

TDRV015_HANDLE   hdl;

/*
** open the specified device
*/
hdl = tdrv015Open(“/dev/tdrv015_0”);
```

```
if (hdl == NULL)
{
   /* handle open error */
}
```

RETURNS

A device handle, or NULL if the function fails. An error code will be stored in *errno*.

ERROR CODES

The error codes are stored in *errno.*

The error code is a standard error code set by the I/O system.

3.1.2 tdrv015Close

NAME

tdrv015Close – close a device.

SYNOPSIS

```
TDRV015_STATUS tdrv015Close
(
      TDRV015_HANDLE        hdl
)
```

DESCRIPTION

This function closes a previously opened device.

PARAMETERS

hdl

This value specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

EXAMPLE

```
#include “tdrv015api.h”

TDRV015_HANDLE   hdl;
TDRV015_STATUS   result;

/*
** close the device
*/
result = tdrv015Close(hdl);
if (result != TDRV015_OK)
{
   /* handle close error */
}
```

RETURNS

On success, TDRV015_OK is returned. In the case of an error, the appropriate error code is returned by the function.

ERROR CODES

Error Code	Description
TDRV015_ERR_INVALID_HANDLE	The specified device handle is invalid

3.1.3 tdrv015GetPciInfo

NAME

tdrv015GetPciInfo – get information of the module PCI header

SYNOPSIS

```
TDRV015_STATUS tdrv015GetPciInfo
(
    TDRV015_HANDLE          hdl,
    TDRV015_PCIINFO_BUF     *pPciInfoBuf
)
```

DESCRIPTION

This function returns information of the module PCI header in the provided data buffer.

PARAMETERS

hdl

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

pPciInfoBuf

This argument is a pointer to the structure TDRV015_PCIINFO_BUF that receives information of the module PCI header.

```
typedef struct
{
    unsigned short      vendorId;
    unsigned short      deviceId;
    unsigned short      subSystemId;
    unsigned short      subSystemVendorId;
    int                 pciBusNo;
    int                 pciDevNo;
    int                 pciFuncNo;
} TDRV015_PCIINFO_BUF;
```

vendorId

PCI module vendor ID.

deviceId

PCI module device ID

subSystemId

PCI module sub system ID

subSystemVendorId

PCI module sub system vendor ID

pciBusNo

Number of the PCI bus, where the module resides.

pciDevNo

PCI device number

pciFuncNo

PCI function number

EXAMPLE

```
#include “tdrv015api.h”

TDRV015_HANDLE       hdl;
TDRV015_STATUS       result;
TDRV015_PCIINFO_BUF  pciInfoBuf

/*
** get module PCI information
*/
result = tdrv015GetPciInfo(hdl, &pciInfoBuf);

if (result != TDRV015_OK)
{
  /* handle error */
}
```

RETURN VALUE

On success, TDRV015_OK is returned. In the case of an error, the appropriate error code is returned by the function.

ERROR CODES

Error Code	Description
TDRV015_ERR_INVALID_HANDLE	The specified device handle is invalid

3.2 Device Access Functions

3.2.1 tdrv015Read8

NAME

tdrv015Read8 – read 8-bit values from PCI BAR space

SYNOPSIS

```
TDRV015_STATUS tdrv015Read8
(
    TDRV015_HANDLE      hdl,
    int                 pciResource,
    int                 offset,
    int                 numItems,
    unsigned char       *pData
)
```

DESCRIPTION

This function reads the specified number of items from the PCI BAR space by using single byte (8-bit) accesses.

The register sets of FPGA on-chip bus slave devices and DRAM memory areas can be accessed via addressable data regions in PCI BAR space.

PARAMETERS

hdl

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

pciResource

This parameter specifies the desired PCI Memory resource to be used for this access. In general, a PCI target (PCIe bridge) supports up to six base address registers. Following values are possible:

Value	Description
TDRV015_RES_MEM_1	First found PCI Memory area.
TDRV015_RES_MEM_2	Second found PCI Memory area.
TDRV015_RES_MEM_3	Third found PCI Memory area.
TDRV015_RES_MEM_4	Fourth found PCI Memory area.
TDRV015_RES_MEM_5	Fifth found PCI Memory area.
TDRV015_RES_MEM_6	Sixth found PCI Memory area.

The Base Address Register usage is programmable and can be changed by modifying the PCIe bridge configuration. Therefore the following table is just an example how the PCI Base Address Registers could be used.

PCI Base Address Register	PCI Address-Type	TDRV015 Resource
0	MEM	TDRV015_RES_MEM_1
1	MEM *(not used)*	TDRV015_RES_MEM_2
2	MEM *(not used)*	TDRV015_RES_MEM_3

offset

This argument specifies the start offset within the PCI BAR space.

numItems

This argument specifies the number of items (8-bit) to read.

pData

This argument is a pointer to an unsigned char buffer which will be filled with the specified number of items from the PCI BAR space. The allocated space must be large enough to hold the specified amount of data.

EXAMPLE

```
#include “tdrv015api.h”

#define NUM 0x10000

TDRV015_HANDLE   hdl;
TDRV015_STATUS   result;
int              offset;
unsigned char    dataBuf[NUM];


offset = 0x30000;
/*
** read 64KB from the DDRA memory page
*/
result = tdrv015Read8(hdl, TDRV015_RES_MEM_1, offset, NUM, dataBuf);

if (result != TDRV015_OK)
{
  /* handle error */
}
```

RETURN VALUE

On success, TDRV015_OK is returned. In the case of an error, the appropriate error code is returned by the function.

ERROR CODES

Error code	Description
TDRV015_ERR_INVALID_HANDLE	The specified device handle is invalid
TDRV015_ERR_INVAL	The specified access range exceeds PCI BAR limits
TDRV015_ERR_ACCESS	The specified PCI resource is not available

3.2.2 tdrv015ReadBE16

NAME

tdrv015ReadBE16 – read 16-bit values from PCI BAR space in big-endian order

SYNOPSIS

```
TDRV015_STATUS tdrv015ReadBE16
(
    TDRV015_HANDLE      hdl,
    int                 pciResource,
    int                 offset,
    int                 numItems,
    unsigned short      *pData
)
```

DESCRIPTION

This function reads the specified number of items from the PCI BAR space by using 16-bit accesses. The values are returned as big-endian values that mean on Intel x86 architectures the multi-byte data will be byte-swapped.

The register sets of FPGA on-chip bus slave devices and DRAM memory areas can be accessed via addressable data regions in PCI BAR space.

PARAMETERS

hdl

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

pciResource

This parameter specifies the desired PCI Memory resource to be used for this access. In general, a PCI target (PCIe bridge) supports up to six base address registers. Following values are possible:

Value	Description
TDRV015_RES_MEM_1	First found PCI Memory area.
TDRV015_RES_MEM_2	Second found PCI Memory area.
TDRV015_RES_MEM_3	Third found PCI Memory area.
TDRV015_RES_MEM_4	Fourth found PCI Memory area.
TDRV015_RES_MEM_5	Fifth found PCI Memory area.
TDRV015_RES_MEM_6	Sixth found PCI Memory area.

The Base Address Register usage is programmable and can be changed by modifying the PCIe bridge configuration. Therefore the following table is just an example how the PCI Base Address Registers could be used.

PCI Base Address Register	PCI Address-Type	TDRV015 Resource
0	MEM	TDRV015_RES_MEM_1
1	MEM *(not used)*	TDRV015_RES_MEM_2
2	MEM *(not used)*	TDRV015_RES_MEM_3

offset

This argument specifies the start offset within the PCI BAR space.

numItems

This argument specifies the number of items (16-bit) to read.

pData

This argument is a pointer to an unsigned short buffer which will be filled with the specified number of items from the PCI BAR space. The allocated space must be large enough to hold the specified amount of data.

EXAMPLE

```
#include “tdrv015api.h”

#define NUM 0x8000

TDRV015_HANDLE   hdl;
TDRV015_STATUS   result;
int              offset;
unsigned short   dataBuf[NUM];

offset = 0x30000;
/*
** read 64KB from the DDRA memory page
*/
result = tdrv015ReadBE16(hdl, TDRV015_RES_MEM_1, offset, NUM, dataBuf);

if (result != TDRV015_OK)
{
  /* handle error */
}
```

RETURN VALUE

On success, TDRV015_OK is returned. In the case of an error, the appropriate error code is returned by the function.

ERROR CODES

Error code	Description
TDRV015_ERR_INVALID_HANDLE	The specified device handle is invalid
TDRV015_ERR_INVAL	The specified access range exceeds PCI BAR limits
TDRV015_ERR_ACCESS	The specified PCI resource is not available

3.2.3 tdrv015ReadLE16

NAME

tdrv015ReadLE16 – read 16-bit values from PCI BAR space in little-endian order

SYNOPSIS

```
TDRV015_STATUS tdrv015ReadLE16
(
    TDRV015_HANDLE      hdl,
    int                 pciResource,
    int                 offset,
    int                 numItems,
    unsigned short      *pData
)
```

DESCRIPTION

This function reads the specified number of items from the PCI BAR space by using 16-bit accesses. The values are returned as little-endian values that means on Intel x86 architectures the multi-byte data will not be byte-swapped.

The register sets of FPGA on-chip bus slave devices and DRAM memory areas can be accessed via addressable data regions in PCI BAR space.

PARAMETERS

hdl

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

pciResource

This parameter specifies the desired PCI Memory resource to be used for this access. In general, a PCI target (PCIe bridge) supports up to six base address registers. Following values are possible:

Value	Description
TDRV015_RES_MEM_1	First found PCI Memory area.
TDRV015_RES_MEM_2	Second found PCI Memory area.
TDRV015_RES_MEM_3	Third found PCI Memory area.
TDRV015_RES_MEM_4	Fourth found PCI Memory area.
TDRV015_RES_MEM_5	Fifth found PCI Memory area.
TDRV015_RES_MEM_6	Sixth found PCI Memory area.

The Base Address Register usage is programmable and can be changed by modifying the PCIe bridge configuration. Therefore the following table is just an example how the PCI Base Address Registers could be used.

PCI Base Address Register	PCI Address-Type	TDRV015 Resource
0	MEM	TDRV015_RES_MEM_1
1	MEM *(not used)*	TDRV015_RES_MEM_2
2	MEM *(not used)*	TDRV015_RES_MEM_3

offset

This argument specifies the start offset within the PCI BAR space.

numItems

This argument specifies the number of items (16-bit) to read.

pData

This argument is a pointer to an unsigned short buffer which will be filled with the specified number of items from the PCI BAR space. The allocated space must be large enough to hold the specified amount of data.

EXAMPLE

```
#include “tdrv015api.h”

#define NUM 0x8000

TDRV015_HANDLE   hdl;
TDRV015_STATUS   result;
int              offset;
unsigned short   dataBuf[NUM];

offset = 0x30000;
/*
** read 64KB from the DDRA memory page
*/
result = tdrv015ReadLE16(hdl, TDRV015_RES_MEM_1, offset, NUM, dataBuf);

if (result != TDRV015_OK)
{
  /* handle error */
}
```

RETURN VALUE

On success, TDRV015_OK is returned. In the case of an error, the appropriate error code is returned by the function.

ERROR CODES

Error Code	Description
TDRV015_ERR_INVALID_HANDLE	The specified device handle is invalid
TDRV015_ERR_INVAL	The specified access range exceeds PCI BAR limits
TDRV015_ERR_ACCESS	The specified PCI resource is not available

3.2.4 tdrv015ReadBE32

NAME

tdrv015ReadBE32 – read 32-bit values from PCI BAR space in big-endian order

SYNOPSIS

```
TDRV015_STATUS tdrv015ReadBE32
(
    TDRV015_HANDLE      hdl,
    int                 pciResource,
    int                 offset,
    int                 numItems,
    unsigned int        *pData
)
```

DESCRIPTION

This function reads the specified number of items from the PCI BAR space by using 32-bit accesses. The values are returned as big-endian values that means on Intel x86 architectures the multi-byte data will be byte-swapped.

The register sets of FPGA on-chip bus slave devices and DRAM memory areas can be accessed via addressable data regions in PCI BAR space.

PARAMETERS

hdl

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

pciResource

This parameter specifies the desired PCI Memory resource to be used for this access. In general, a PCI target (PCIe bridge) supports up to six base address registers. Following values are possible:

Value	Description
TDRV015_RES_MEM_1	First found PCI Memory area.
TDRV015_RES_MEM_2	Second found PCI Memory area.
TDRV015_RES_MEM_3	Third found PCI Memory area.
TDRV015_RES_MEM_4	Fourth found PCI Memory area.
TDRV015_RES_MEM_5	Fifth found PCI Memory area.
TDRV015_RES_MEM_6	Sixth found PCI Memory area.

The Base Address Register usage is programmable and can be changed by modifying the PCIe bridge configuration. Therefore the following table is just an example how the PCI Base Address Registers could be used.

PCI Base Address Register	PCI Address-Type	TDRV015 Resource
0	MEM	TDRV015_RES_MEM_1
1	MEM *(not used)*	TDRV015_RES_MEM_2
2	MEM *(not used)*	TDRV015_RES_MEM_3

offset

This argument specifies the start offset within the PCI BAR space.

numItems

This argument specifies the number of items (32-bit) to read.

pData

This argument is a pointer to an unsigned int buffer which will be filled with the specified number of items from the PCI BAR space. The allocated space must be large enough to hold the specified amount of data.

EXAMPLE

```
#include “tdrv015api.h”

#define NUM 4

TDRV015_HANDLE   hdl;
TDRV015_STATUS   result;
int              offset;
unsigned int     dataBuf[NUM];

offset = 0;
/*
** read all registers from the control component
*/
result = tdrv015ReadBE32(hdl, TDRV015_RES_MEM_1, offset, NUM, dataBuf);

if (result != TDRV015_OK)
{
  /* handle error */
}
```

RETURN VALUE

On success, TDRV015_OK is returned. In the case of an error, the appropriate error code is returned by the function.

ERROR CODES

Error Code	Description
TDRV015_ERR_INVALID_HANDLE	The specified device handle is invalid
TDRV015_ERR_INVAL	The specified access range exceeds PCI BAR limits
TDRV015_ERR_ACCESS	The specified PCI resource is not available

3.2.5 tdrv015ReadLE32

NAME

tdrv015ReadLE32 – read 32-bit values from PCI BAR space in little-endian order

SYNOPSIS

```
TDRV015_STATUS tdrv015ReadLE32
(
    TDRV015_HANDLE      hdl,
    int                 pciResource,
    int                 offset,
    int                 numItems,
    unsigned int        *pData
)
```

DESCRIPTION

This function reads the specified number of items from the PCI BAR space by using 32-bit accesses. The values are returned as little-endian values that means on Intel x86 architectures the multi-byte data will not be byte-swapped.

The register sets of FPGA on-chip bus slave devices and DRAM memory areas can be accessed via addressable data regions in PCI BAR space.

PARAMETERS

hdl

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

pciResource

This parameter specifies the desired PCI Memory resource to be used for this access. In general, a PCI target (PCIe bridge) supports up to six base address registers. Following values are possible:

Value	Description
TDRV015_RES_MEM_1	First found PCI Memory area.
TDRV015_RES_MEM_2	Second found PCI Memory area.
TDRV015_RES_MEM_3	Third found PCI Memory area.
TDRV015_RES_MEM_4	Fourth found PCI Memory area.
TDRV015_RES_MEM_5	Fifth found PCI Memory area.
TDRV015_RES_MEM_6	Sixth found PCI Memory area.

The Base Address Register usage is programmable and can be changed by modifying the PCIe bridge configuration. Therefore the following table is just an example how the PCI Base Address Registers could be used.

PCI Base Address Register	PCI Address-Type	TDRV015 Resource
0	MEM	TDRV015_RES_MEM_1
1	MEM *(not used)*	TDRV015_RES_MEM_2
2	MEM *(not used)*	TDRV015_RES_MEM_3

offset

This argument specifies the start offset within the PCI BAR space.

numItems

This argument specifies the number of items (32-bit) to read.

pData

This argument is a pointer to an unsigned int buffer which will be filled with the specified number of items from the PCI BAR space. The allocated space must be large enough to hold the specified amount of data.

EXAMPLE

```
#include “tdrv015api.h”

#define NUM   4

TDRV015_HANDLE   hdl;
TDRV015_STATUS   result;
int              offset;
unsigned int     dataBuf[NUM];

offset = 0;
/*
** read all registers from the control component
*/
result = tdrv015ReadLE32(hdl, TDRV015_RES_MEM_1, offset, NUM, dataBuf);

if (result != TDRV015_OK)
{
  /* handle error */
}
```

RETURN VALUE

On success, TDRV015_OK is returned. In the case of an error, the appropriate error code is returned by the function.

ERROR CODES

Error Code	Description
TDRV015_ERR_INVALID_HANDLE	The specified device handle is invalid
TDRV015_ERR_INVAL	The specified access range exceeds PCI BAR limits
TDRV015_ERR_ACCESS	The specified PCI resource is not available

3.2.6 tdrv015Write8

NAME

tdrv015Write8 – write 8-bit values to the PCI BAR space

SYNOPSIS

```
TDRV015_STATUS tdrv015Write8
(
    TDRV015_HANDLE      hdl,
    int                 pciResource,
    int                 offset,
    int                 numItems,
    unsigned char       *pData
)
```

DESCRIPTION

This function writes the specified number of items to the PCI BAR space by using single byte (8-bit) accesses.

The register sets of FPGA on-chip bus slave devices and DRAM memory areas can be accessed via addressable data regions in PCI BAR space.

PARAMETERS

hdl

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

pciResource

This parameter specifies the desired PCI Memory resource to be used for this access. In general, a PCI target (PCIe bridge) supports up to six base address registers. Following values are possible:

Value	Description
TDRV015_RES_MEM_1	First found PCI Memory area.
TDRV015_RES_MEM_2	Second found PCI Memory area.
TDRV015_RES_MEM_3	Third found PCI Memory area.
TDRV015_RES_MEM_4	Fourth found PCI Memory area.
TDRV015_RES_MEM_5	Fifth found PCI Memory area.
TDRV015_RES_MEM_6	Sixth found PCI Memory area.

The Base Address Register usage is programmable and can be changed by modifying the PCIe bridge configuration. Therefore the following table is just an example how the PCI Base Address Registers could be used.

PCI Base Address Register	PCI Address-Type	TDRV015 Resource
0	MEM	TDRV015_RES_MEM_1
1	MEM *(not used)*	TDRV015_RES_MEM_2
2	MEM *(not used)*	TDRV015_RES_MEM_3

offset

This argument specifies the start offset within the PCI BAR space.

numItems

This argument specifies the number of items (8-bit) to write.

pData

This argument is a pointer to an unsigned char buffer with the data items to write. The allocated space must be large enough to hold the specified amount of data.

EXAMPLE

```
#include “tdrv015api.h”

#define NUM 0x10000

TDRV015_HANDLE   hdl;
TDRV015_STATUS   result;
int              offset;
unsigned char    dataBuf[NUM];

dataBuf[0] = 0xAA;
dataBuf[1] = 0x55;
…

offset = 0x40000;

/*
** write 64KB to the DDRB memory page
*/
result = tdrv015Write8(hdl, TDRV015_RES_MEM_1, offset, NUM, dataBuf);

if (result != TDRV015_OK)
{
  /* handle error */
}
```

RETURN VALUE

On success, TDRV015_OK is returned. In the case of an error, the appropriate error code is returned by the function.

ERROR CODES

Error Code	Description
TDRV015_ERR_INVALID_HANDLE	The specified device handle is invalid
TDRV015_ERR_INVAL	The specified access range exceeds PCI BAR limits
TDRV015_ERR_ACCESS	The specified PCI resource is not available

3.2.7 tdrv015WriteBE16

NAME

tdrv015WriteBE16 – write 16-bit values to the PCI BAR space big-endian order

SYNOPSIS

```
TDRV015_STATUS tdrv015WriteBE16
(
    TDRV015_HANDLE      hdl,
    int                 pciResource,
    int                 offset,
    int                 numItems,
    unsigned short      *pData
)
```

DESCRIPTION

This function writes the specified number of items to the PCI BAR space by using 16-bit accesses.

The values are written in big-endian order that means on Intel x86 architectures the multi-byte data will be byte-swapped.

The register sets of FPGA on-chip bus slave devices and DRAM memory areas can be accessed via addressable data regions in PCI BAR space.

PARAMETERS

hdl

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

pciResource

This parameter specifies the desired PCI Memory resource to be used for this access. In general, a PCI target (PCIe bridge) supports up to six base address registers. Following values are possible:

Value	Description
TDRV015_RES_MEM_1	First found PCI Memory area.
TDRV015_RES_MEM_2	Second found PCI Memory area.
TDRV015_RES_MEM_3	Third found PCI Memory area.
TDRV015_RES_MEM_4	Fourth found PCI Memory area.
TDRV015_RES_MEM_5	Fifth found PCI Memory area.
TDRV015_RES_MEM_6	Sixth found PCI Memory area.

The Base Address Register usage is programmable and can be changed by modifying the PCIe bridge configuration. Therefore the following table is just an example how the PCI Base Address Registers could be used.

PCI Base Address Register	PCI Address-Type	TDRV015 Resource
0	MEM	TDRV015_RES_MEM_1
1	MEM *(not used)*	TDRV015_RES_MEM_2
2	MEM *(not used)*	TDRV015_RES_MEM_3

offset

This argument specifies the start offset within the PCI BAR space.

numItems

This argument specifies the number of items (16-bit) to write.

pData

This argument is a pointer to an unsigned short buffer with the data items to write. The allocated space must be large enough to hold the specified amount of data.

EXAMPLE

```
#include “tdrv015api.h”

#define NUM 0x8000

TDRV015_HANDLE   hdl;
TDRV015_STATUS   result;
int              offset;
unsigned short   dataBuf[NUM];

dataBuf[0] = 0xAA55;
dataBuf[1] = 0x55AA;
…

offset = 0x30000;
/*
** write 64KB to the DDRA memory page
*/
result = tdrv015WriteBE16(hdl, TDRV015_RES_MEM_1, offset, NUM, dataBuf);

if (result != TDRV015_OK)
{
  /* handle error */
}
```

RETURN VALUE

On success, TDRV015_OK is returned. In the case of an error, the appropriate error code is returned by the function.

ERROR CODES

Error Code	Description
TDRV015_ERR_INVALID_HANDLE	The specified device handle is invalid
TDRV015_ERR_INVAL	The specified access range exceeds PCI BAR limits
TDRV015_ERR_ACCESS	The specified PCI resource is not available

3.2.8 tdrv015WriteLE16

NAME

tdrv015WriteLE16 – write 16-bit values to the PCI BAR space in little-endian order

SYNOPSIS

```
TDRV015_STATUS tdrv015WriteLE16
(
    TDRV015_HANDLE      hdl,
    int                 pciResource,
    int                 offset,
    int                 numItems,
    unsigned short      *pData
)
```

DESCRIPTION

This function writes the specified number of items to the PCI BAR space by using 16-bit accesses.

The values are written in little-endian order that means on Intel x86 architectures the multi-byte data will not be byte-swapped.

The register sets of FPGA on-chip bus slave devices and DRAM memory areas can be accessed via addressable data regions in PCI BAR space.

PARAMETERS

hdl

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

pciResource

This parameter specifies the desired PCI Memory resource to be used for this access. In general, a PCI target (PCIe bridge) supports up to six base address registers. Following values are possible:

Value	Description
TDRV015_RES_MEM_1	First found PCI Memory area.
TDRV015_RES_MEM_2	Second found PCI Memory area.
TDRV015_RES_MEM_3	Third found PCI Memory area.
TDRV015_RES_MEM_4	Fourth found PCI Memory area.
TDRV015_RES_MEM_5	Fifth found PCI Memory area.
TDRV015_RES_MEM_6	Sixth found PCI Memory area.

The Base Address Register usage is programmable and can be changed by modifying the PCIe bridge configuration. Therefore the following table is just an example how the PCI Base Address Registers could be used.

PCI Base Address Register	PCI Address-Type	TDRV015 Resource
0	MEM	TDRV015_RES_MEM_1
1	MEM *(not used)*	TDRV015_RES_MEM_2
2	MEM *(not used)*	TDRV015_RES_MEM_3

offset

This argument specifies the start offset within the PCI BAR space.

numItems

This argument specifies the number of items (16-bit) to write.

pData

This argument is a pointer to an unsigned short buffer with the data items to write. The allocated space must be large enough to hold the specified amount of data.

EXAMPLE

```
#include “tdrv015api.h”

#define NUM   0x8000

TDRV015_HANDLE   hdl;
TDRV015_STATUS   result;
int              offset;
unsigned short   dataBuf[NUM];

dataBuf[0] = 0xAA55;
dataBuf[1] = 0x55AA;
…

offset = 0x30000;
/*
** write 64KB to the DDRA memory page
*/
result = tdrv015WriteLE16(hdl, TDRV015_RES_MEM_1, offset, NUM, dataBuf);

if (result != TDRV015_OK)
{
  /* handle error */
}
```

RETURN VALUE

On success, TDRV015_OK is returned. In the case of an error, the appropriate error code is returned by the function.

ERROR CODES

Error Code	Description
TDRV015_ERR_INVALID_HANDLE	The specified device handle is invalid
TDRV015_ERR_INVAL	The specified access range exceeds PCI BAR limits
TDRV015_ERR_ACCESS	The specified PCI resource is not available

3.2.9 tdrv015WriteBE32

NAME

tdrv015WriteBE32 – write 32-bit values to the PCI BAR space big-endian order

SYNOPSIS

```
TDRV015_STATUS tdrv015WriteBE32
(
    TDRV015_HANDLE      hdl,
    int                 pciResource,
    int                 offset,
    int                 numItems,
    unsigned short      *pData
)
```

DESCRIPTION

This function writes the specified number of items to the PCI BAR space by using 32-bit accesses.

The values are written in big-endian order that means on Intel x86 architectures the multi-byte data will be byte-swapped.

The register sets of FPGA on-chip bus slave devices and DRAM memory areas can be accessed via addressable data regions in PCI BAR space.

PARAMETERS

hdl

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

pciResource

This parameter specifies the desired PCI Memory resource to be used for this access. In general, a PCI target (PCIe bridge) supports up to six base address registers. Following values are possible:

Value	Description
TDRV015_RES_MEM_1	First found PCI Memory area.
TDRV015_RES_MEM_2	Second found PCI Memory area.
TDRV015_RES_MEM_3	Third found PCI Memory area.
TDRV015_RES_MEM_4	Fourth found PCI Memory area.
TDRV015_RES_MEM_5	Fifth found PCI Memory area.
TDRV015_RES_MEM_6	Sixth found PCI Memory area.

The Base Address Register usage is programmable and can be changed by modifying the PCIe bridge configuration. Therefore the following table is just an example how the PCI Base Address Registers could be used.

PCI Base Address Register	PCI Address-Type	TDRV015 Resource
0	MEM	TDRV015_RES_MEM_1
1	MEM *(not used)*	TDRV015_RES_MEM_2
2	MEM *(not used)*	TDRV015_RES_MEM_3

offset

This argument specifies the start offset within the PCI BAR space.

numItems

This argument specifies the number of items (32-bit) to write.

pData

This argument is a pointer to an unsigned short buffer with the data items to write. The allocated space must be large enough to hold the specified amount of data.

EXAMPLE

```
#include “tdrv015api.h”

TDRV015_HANDLE   hdl;
TDRV015_STATUS   result;
int              offset;
unsigned int     data;

data   = 0x10020000;    /* PLB DDRA address space */
offset = 0x20004;    /* memory controller device register */
/*
** adjust the selected memory page from DDRA to 0x10020000 by
** setting the DDRA memory address register in the memory controller
** device
*/
result = tdrv015WriteBE32(hdl, TDRV015_RES_MEM_1, offset, 1, &data);

if (result != TDRV015_OK)
{
  /* handle error */
}
```

RETURN VALUE

On success, TDRV015_OK is returned. In the case of an error, the appropriate error code is returned by the function.

ERROR CODES

Error Code	Description
TDRV015_ERR_INVALID_HANDLE	The specified device handle is invalid
TDRV015_ERR_INVAL	The specified access range exceeds PCI BAR limits
TDRV015_ERR_ACCESS	The specified PCI resource is not available

3.2.10 tdrv015WriteLE32

NAME

tdrv015WriteLE32 – write 32-bit values to the PCI BAR space in little-endian order

SYNOPSIS

```
TDRV015_STATUS tdrv015WriteLE32
(
    TDRV015_HANDLE      hdl,
    int                 pciResource,
    int                 offset,
    int                 numItems,
    unsigned short      *pData
)
```

DESCRIPTION

This function writes the specified number of items to the PCI BAR space by using 32-bit accesses.

The values are written in little-endian order that means on Intel x86 architectures the multi-byte data will not be byte-swapped.

The register sets of FPGA on-chip bus slave devices and DRAM memory areas can be accessed via addressable data regions in PCI BAR space.

PARAMETERS

hdl

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

pciResource

This parameter specifies the desired PCI Memory resource to be used for this access. In general, a PCI target (PCIe bridge) supports up to six base address registers. Following values are possible:

Value	Description
TDRV015_RES_MEM_1	First found PCI Memory area.
TDRV015_RES_MEM_2	Second found PCI Memory area.
TDRV015_RES_MEM_3	Third found PCI Memory area.
TDRV015_RES_MEM_4	Fourth found PCI Memory area.
TDRV015_RES_MEM_5	Fifth found PCI Memory area.
TDRV015_RES_MEM_6	Sixth found PCI Memory area.

The Base Address Register usage is programmable and can be changed by modifying the PCIe bridge configuration. Therefore the following table is just an example how the PCI Base Address Registers could be used.

PCI Base Address Register	PCI Address-Type	TDRV015 Resource
0	MEM	TDRV015_RES_MEM_1
1	MEM *(not used)*	TDRV015_RES_MEM_2
2	MEM *(not used)*	TDRV015_RES_MEM_3

offset

This argument specifies the start offset within the PCI BAR space.

numItems

This argument specifies the number of items (32-bit) to write.

pData

This argument is a pointer to an unsigned short buffer with the data items to write. The allocated space must be large enough to hold the specified amount of data.

EXAMPLE

```
#include “tdrv015api.h”

TDRV015_HANDLE   hdl;
TDRV015_STATUS   result;
int              offset;
unsigned int     data;

data   = 0x10020000;    /* PLB DDRA address space */
offset = 0x20004;    /* memory controller device register */
/*
** adjust the selected memory page from DDRA to 0x10020000 by
** setting the DDRA memory address register in the memory controller
** device
*/
result = tdrv015WriteLE32(hdl, TDRV015_RES_MEM_1, offset, 1, &data);

if (result != TDRV015_OK)
{
  /* handle error */
}
```

RETURN VALUE

On success, TDRV015_OK is returned. In the case of an error, the appropriate error code is returned by the function.

ERROR CODES

Error Code	Description
TDRV015_ERR_INVALID_HANDLE	The specified device handle is invalid
TDRV015_ERR_INVAL	The specified access range exceeds PCI BAR limits
TDRV015_ERR_ACCESS	The specified PCI resource is not available

3.3 Resource Mapping Functions

3.3.1 tdrv015PciResourceMap

NAME

tdrv015PciResourceMap – map a PCI resource directly into the process context

SYNOPSIS

```
TDRV015_STATUS tdrv015PciResourceMap
(
    TDRV015_HANDLE      hdl,
    int                 pciResource,
    unsigned char       **pPtr,
    unsigned int        *pSize
)
```

DESCRIPTION

This function maps the specified PCI resource of the hardware module directly into the process context. The retrieved pointer can be used for direct non-cached register access.

PARAMETERS

hdl

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

pciResource

This parameter specifies the desired PCI Memory resource to be used for this access. In general, a PCI target (PCIe bridge) supports up to six base address registers. Following values are possible:

Value	Description
TDRV015_RES_MEM_1	First found PCI Memory area.
TDRV015_RES_MEM_2	Second found PCI Memory area.
TDRV015_RES_MEM_3	Third found PCI Memory area.
TDRV015_RES_MEM_4	Fourth found PCI Memory area.
TDRV015_RES_MEM_5	Fifth found PCI Memory area.
TDRV015_RES_MEM_6	Sixth found PCI Memory area.

The Base Address Register usage is programmable and can be changed by modifying the PCIe bridge configuration. Therefore the following table is just an example how the PCI Base Address Registers could be used.

PCI Base Address Register	PCI Address-Type	TDRV015 Resource
0	MEM	TDRV015_RES_MEM_1
1	MEM *(not used)*	TDRV015_RES_MEM_2
2	MEM *(not used)*	TDRV015_RES_MEM_3

pPtr

This argument is a pointer to an unsigned char pointer that receives the start address of the mapped PCI resource.

pSize

This argument returns the size of the mapped PCI resource in bytes.

EXAMPLE

```
#include “tdrv015api.h”

TDRV015_HANDLE   hdl;
TDRV015_STATUS   result;
unsigned char    *pReg;
unsigned int     size;

/*
** map first memory PCI resource
*/
result = tdrv015PciResourceMap(hdl, TDRV015_RES_MEM_1, &pReg, &size);

if (result != TDRV015_OK)
{
  /* handle error */
}
```

RETURN VALUE

On success, TDRV015_OK is returned. In the case of an error, the appropriate error code is returned by the function.

ERROR CODES

Error Code	Description
TDRV015_ERR_INVALID_HANDLE	The specified device handle is invalid
TDRV015_ERR_ACCESS	Specified PCI resource not available
TDRV015_ERR_NOMEM	Unable to allocate memory

3.3.2 tdrv015PciResourceUnmap

NAME

tdrv015PciResourceUnmap – unmap a previously mapped PCI resource

SYNOPSIS

```
TDRV015_STATUS tdrv015PciResourceUnmap
(
    TDRV015_HANDLE      hdl,
    unsigned char       *pPtr
)
```

DESCRIPTION

This function unmaps a previously mapped PCI resource, freeing the system resources used for this mapping.

PARAMETERS

hdl

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

pPtr

This argument is a pointer to an unsigned char pointer that represents the start address of the previously mapped PCI resource. This pointer must have been received from the corresponding mapping function.

EXAMPLE

```
#include “tdrv015api.h”

TDRV015_HANDLE   hdl;
TDRV015_STATUS   result;
unsigned char    *pReg;

/*
** unmap a previously mapped PCI resource
*/
result = tdrv015PciResourceUnmap(hdl, pReg);

if (result != TDRV015_OK)
{
  /* handle error */
}
```

RETURN VALUE

On success, TDRV015_OK is returned. In the case of an error, the appropriate error code is returned by the function.

ERROR CODES

Error Code	Description
TDRV015_ERR_INVALID_HANDLE	The specified device handle is invalid
TDRV015_ERR_INVAL	Invalid pointer specified

3.4 DMA Functions

3.4.1 tdrv015DmaFpgaToHost

NAME

tdrv015DmaFpgaToHost – initiate DMA transfer from FPGA on-chip bus to host PCI bus

SYNOPSIS

```
TDRV015_STATUS tdrv015DmaFpgaToHost
(
    TDRV015_HANDLE      hdl,
    unsigned char       *pSrcAddr,
    unsigned char       *pDestAddr,
    unsigned int        numBytes
    int                 timeout
)
```

DESCRIPTION

This function initiates a DMA transfer from a FPGA on-chip bus resource (e.g. SDRAM banks) to the PCI bus of the host system by using the DMA controller of the infrastructure module. Depending on the timeout parameter the function will be blocked until the transfer has finished or the function returns immediately after starting the transfer.

PARAMETERS

hdl

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

pSrcAddr

This argument specifies a valid 32-bit DMA source start address in the FPGA on-chip bus space (e.g. SDRAM bank).

pDestAddr

This argument specifies a valid 32-bit DMA destination start address in the host PCI bus space (e.g. host RAM space). The destination address must be a valid PCI bus address. To allocate a DMA buffer in the host RAM the function tdrv015GetDmaBuf should be used.

numBytes

This argument specifies the number of bytes to transfer.

timeout

This value specifies the timeout (in milliseconds) the user is willing to wait for this operation to complete. The granularity depends on the used operating system. Specify 0 to return immediately after initiating the DMA transfer, without aborting it. Specify -1 to wait indefinitely for completion.

EXAMPLE

```
#include “tdrv015api.h”

TDRV015_HANDLE   hdl;
TDRV015_STATUS   result;
unsigned char    *pBuf, *pPlb;

/*
** allocate a 64KB DMA buffer in host RAM
*/
result = tdrv015GetDmaBuf(hdl, 0x10000, &pBuf);

if (result != TDRV015_OK)
{
  /* handle error */
}

/*
** transfer 64KB data from PLB DDRB to a buffer in host RAM.
** Wait at least 1 second for completion
*/
pPlb = (unsigned char*)0x20000000;

result = tdrv015DmaFpgaToHost(hdl, pPlb, pBuf, 0x10000, 1000);

if (result != TDRV015_OK)
{
  /* handle error */
}
```

RETURN VALUE

On success, TDRV015_OK is returned. In the case of an error, the appropriate error code is returned by the function.

ERROR CODES

Error Code	Description
TDRV015_ERR_INVALID_HANDLE	The specified device handle is invalid
TDRV015_ERR_INVAL	Not a valid SDRAM address range
TDRV015_ERR_BUSY	Unable to gain exclusive access to the DMA controller within the specified time.
TDRV015_ERR_NOMEM	Unable to allocate memory for DMA descriptor list
TDRV015_ERR_IO	A DMA error has occurred
TDRV015_ERR_TIMEOUT	The specified timeout occurred

3.4.2 tdrv015DmaHostToFpga

NAME

tdrv015DmaHostToFpga – initiate DMA transfer from PCI bus to FPGA on-chip bus

SYNOPSIS

```
TDRV015_STATUS tdrv015DmaHostToFpga
(
    TDRV015_HANDLE      hdl,
    unsigned char       *pSrcAddr,
    unsigned char       *pDestAddr,
    unsigned int        numBytes
    int                 timeout
)
```

DESCRIPTION

This function initiates a DMA transfer from the PCI bus of the host system to a FPGA on-chip bus resource (e.g SDRAM banks) by using the DMA controller of the infrastructure module. Depending on the timeout parameter the function will be blocked until the transfer has finished or the function returns immediately after starting the transfer.

PARAMETERS

hdl

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

pSrcAddr

This argument specifies a valid 32-bit DMA source start address in the host PCI bus space (e.g. host RAM space). The destination address must be a valid PCI bus address. To allocate a DMA buffer in the host RAM the function tdrv015GetDmaBuf should be used.

pDestAddr

This argument specifies a valid 32-bit DMA destination start address in the FPGA on-chip bus space (e.g. SDRAM bank).

numBytes

This argument specifies the number of bytes to transfer.

timeout

This value specifies the timeout (in milliseconds) the user is willing to wait for this operation to complete. The granularity depends on the used operating system. Specify 0 to return immediately after initiating the DMA transfer, without aborting it. Specify -1 to wait indefinitely for completion.

EXAMPLE

```
#include “tdrv015api.h”

TDRV015_HANDLE   hdl;
TDRV015_STATUS   result;
unsigned char    *pBuf, *pPlb;

/*
** allocate a 2MB DMA buffer in host RAM
*/
result = tdrv015GetDmaBuf(hdl, 0x00200000, &pBuf);
if (result != TDRV015_OK)
{
  /* handle error */
}

/* fill data buffer */

/*
** transfer 2MB data from host RAM to PLB DDRA
** Wait up to 10 seconds for completion.
*/
pPlb = (unsigned char*)0x10000000;
result = tdrv015DmaHostToFpga(hdl, pBuf, pPlb, 0x00200000, 10000);

if (result != TDRV015_OK)
{
  /* handle error */
}
```

RETURN VALUE

On success, TDRV015_OK is returned. In the case of an error, the appropriate error code is returned by the function.

ERROR CODES

Error Code	Description
TDRV015_ERR_INVALID_HANDLE	The specified device handle is invalid
TDRV015_ERR_INVAL	Not a valid SDRAM address range
TDRV015_ERR_BUSY	Unable to gain exclusive access to the DMA controller within the specified time.
TDRV015_ERR_NOMEM	Unable to allocate memory for DMA descriptor list
TDRV015_ERR_IO	A DMA error has occurred
TDRV015_ERR_TIMEOUT	The specified timeout occurred

3.4.3 tdrv015DmaGetStatus

NAME

tdrv015DmaGetStatus – read the status of the DMA controller

SYNOPSIS

```
TDRV015_STATUS tdrv015DmaGetStatus
(
    TDRV015_HANDLE      hdl,
    unsigned int        *pDmaStatus
)
```

DESCRIPTION

This function returns the status of the DMA controller, namely if a transfer is in progress or if an error has occurred. This function can be used to check if a DMA transfer has already been completed. An occurred error is cleared after execution of this function.

PARAMETERS

hdl

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

pDmaStatus

This argument is a pointer to an *unsigned int* buffer, where the status of the DMA controller is returned. Possible values are:

Value	Description
TDRV015_DMASTAT_READY	DMA controller is ready for a new transfer.
TDRV015_DMASTAT_BUSY	DMA controller is busy with a transfer.
TDRV015_DMASTAT_ERROR	An error has occurred during the last DMA transfer.

EXAMPLE

```
#include “tdrv015api.h”

TDRV015_HANDLE   hdl;
TDRV015_STATUS   result;
unsigned int     dmaStatus;

/*
** get status of DMA controller
*/
result = tdrv015DmaGetStatus(hdl, &dmaStatus);

if (result != TDRV015_OK)
{
  /* handle error */
}
```

RETURN VALUE

On success, TDRV015_OK is returned. In the case of an error, the appropriate error code is returned by the function.

ERROR CODES

Error Code	Description
TDRV015_ERR_INVALID_HANDLE	The specified device handle is invalid

3.4.4 tdrv015DmaStop

NAME

tdrv015DmaStop – abort an active DMA transfer

SYNOPSIS

```
TDRV015_STATUS tdrv015DmaStop
(
      TDRV015_HANDLE              hdl
)
```

DESCRIPTION

This function aborts an active DMA transfer, and causes a reset of the DMA controller.

PARAMETERS

hdl

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

EXAMPLE

```
#include “tdrv015api.h”

TDRV015_HANDLE hdl;
TDRV015_STATUS result;

/*
** stop DMA transfer
*/
result = tdrv015DmaStop(hdl);

if (result != TDRV015_OK)
{
  /* handle error */
}
```

RETURN VALUE

On success, TDRV015_OK is returned. In the case of an error, the appropriate error code is returned by the function.

ERROR CODES

Error Code	Description
TDRV015_ERR_INVALID_HANDLE	The specified device handle is invalid

3.4.5 tdrv015GetDmaBuf

NAME

tdrv015GetDmaBuf – allocate DMA buffer in host RAM

SYNOPSIS

```
TDRV015_STATUS tdrv015GetDmaBuf
(
    TDRV015_HANDLE      hdl,
    unsigned int        size,
    unsigned char       **pPtr
)
```

DESCRIPTION

This function allocates the specified amount of non-cached, PCI bus accessible memory in the Host RAM. The allocated memory can be freed with the function tdrv015FreeDmaBuf. The maximum amount of contiguous DMA memory depends on the specific system environment.

PARAMETERS

hdl

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

size

This argument specifies the number of bytes to allocate.

pPtr

This argument is a pointer to an unsigned char pointer that receives the start address of the DMA buffer

EXAMPLE

```
#include “tdrv015api.h”

TDRV015_HANDLE   hdl;
TDRV015_STATUS   result;
unsigned char    *pBuf;

/*
** allocate a 2MB DMA buffer in host RAM
*/
result = tdrv015GetDmaBuf(hdl, 0x00200000, &pBuf);

if (result != TDRV015_OK)
{
  /* handle error */
}
```

RETURN VALUE

On success, TDRV015_OK is returned. In the case of an error, the appropriate error code is returned by the function.

ERROR CODES

Error Code	Description
TDRV015_ERR_INVALID_HANDLE	The specified device handle is invalid
TDRV015_ERR_NOMEM	Unable to allocate memory

3.4.6 tdrv015FreeDmaBuf

NAME

tdrv015FreeDmaBuf – free previously allocated DMA buffer

SYNOPSIS

```
TDRV015_STATUS tdrv015FreeDmaBuf
(
      TDRV015_HANDLE      hdl,
      unsigned char       *pPtr
)
```

DESCRIPTION

This function frees a DMA buffer which was previously allocated with tdrv015GetDmaBuf.

PARAMETERS

hdl

This argument specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

pPtr

This argument is a pointer to the buffer to free.

EXAMPLE

```
#include “tdrv015api.h”

TDRV015_HANDLE   hdl;
TDRV015_STATUS   result;
unsigned char    *pBuf;

result = tdrv015FreeDmaBuf(hdl, pBuf);

if (result != TDRV015_OK)
{
   /* handle error */
}
```

RETURN VALUE

On success, TDRV015_OK is returned. In the case of an error, the appropriate error code is returned by the function.

ERROR CODES

Error Code	Description
TDRV015_ERR_INVALID_HANDLE	The specified device handle is invalid

3.5 Interrupt Functions

3.5.1 tdrv015InterruptWait

NAME

tdrv015InterruptWait – Wait for incoming Local Interrupt Source

SYNOPSIS

```
TDRV015_STATUS tdrv015InterruptWait
(
    TDRV015_HANDLE      hdl,
    unsigned int        interruptMask,
    unsigned int        *pInterruptOccurred,
    int                 timeout
);
```

DESCRIPTION

This function enables the specified local interrupt sources, and waits for interrupts on the specified local interrupt sources. After an interrupt has arrived, the corresponding occurred local interrupt source is disabled inside the Infrastructure Module (IM). Multiple functions may wait for the same interrupt source to occur.

The delay between an incoming interrupt and the return of the described function is system-dependent, and is most likely several microseconds. If a faster response (lower interrupt latency) to an interrupt request is required an interrupt callback function (Fehler! Verweisquelle konnte nicht gefunden werden.**) should be installed to handle the interrupt.**

PARAMETERS

hdl

This value specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

interruptMask

This parameter specifies specific interrupt bits to wait for. The interrupt bits correspond to the Infrastructure Module's "Interrupt Pending Register" bits described in the FDK user manual. Please refer to the hardware user manual for further information on the possible interrupt bits. The function returns if at least one of the specified interrupt sources is detected.

pInterruptOccurred

If at least one of the specified interrupt sources occurs, the value is returned through this pointer. The interrupt bits correspond to the Infrastructure Module's "Interrupt Pending Register" bits described in the FDK user manual. Please refer to the hardware user manual for further information on the possible interrupt bits.

timeout

This value specifies the timeout in milliseconds the function will wait for the interrupt to arrive. The granularity depends on the operating system. To wait indefinitely, specify -1 as timeout parameter.

EXAMPLE

```
#include "tdrv015api.h"

TDRV015_HANDLE   hdl;
TDRV015_STATUS   result;
unsigned int     interruptMask;
unsigned int     interruptOccurred;

/*
** Wait at least 5 seconds for incoming interrupts on UINTP0 and/or UINTP1
*/
interruptMask = (1 << 17) | (1 << 16);
result = tdrv015InterruptWait(   hdl,
                 interruptMask,
                 &interruptOccurred,
                 5000 );
if (result == TDRV015_OK)
{
   /* Interrupt arrived.                                  */
   /* Now acknowledge interrupt source in FPGA logic   */
   /* to clear the Local Interrupt Source.             */
   /* Use tdrv015Read and tdrv015Write functions for   */
   /* register access.                                    */
} else {
   /* handle error */
}
```

RETURNS

On success, TDRV015_OK is returned. In the case of an error, the appropriate error code is returned by the function.

ERROR CODES

Error Code	Description
TDRV015_ERR_INVALID_HANDLE	The specified TDRV015_HANDLE is invalid.
TDRV015_ERR_NORESOURCE	Failed to allocate resources for the wait job
TDRV015_ERR_TIMEOUT	The specified timeout occurred.

3.5.2 tdrv015InterruptRegisterCallbackThread

NAME

tdrv015InterruptRegisterCallbackThread – Register a User Callback Function for Interrupt Handling

SYNOPSIS

```
TDRV015_STATUS tdrv015InterruptRegisterCallbackThread
(
    TDRV015_HANDLE      hdl,
    int                 threadPriority,
    int                 stackSize,
    unsigned int        interruptMask,
    FUNCINTCALLBACK     callbackFunction,
    void                *funcparam,
    TDRV015_HANDLE      *pCallbackHandle
)
```

DESCRIPTION

This function registers a user callback function which is executed after detection of the specified interrupt source. It is possible to register multiple callback functions to one or a set (bit mask) of interrupt sources.

The callback function is executed in a thread context, so using TDRV015 device driver functions and system functions is allowed. The callback function should be kept as short as possible. The specified callback function is executed with the occurred interrupt bits and the specified function parameter as function arguments. Additionally, a status value is passed to the callback function, which reflects the result of the involved API functions.

The delay between an incoming interrupt and the execution of the callback function is system-dependent, and is most likely several microseconds.

PARAMETERS

hdl

This value specifies the device handle to the hardware module retrieved by a call to the corresponding open-function.

threadPriority

This parameter specifies the priority to be used for the callback thread. Possible values are:

Value	Description
TDRV015_PRIORITY_NORMAL	Normal thread priority (0)
TDRV015_PRIORITY_HIGH	Higher thread priority (-1)
TDRV015_PRIORITY_LOW	Lower thread priority (1)

Other values might be possible (see also Linux PTHREAD documentation).

stackSize

This parameter specifies the stack size to be used for the callback thread. The value is specified in bytes.

interruptMask

This parameter specifies specific interrupt bits to wait for. The interrupt bits correspond to the Infrastructure Module's "Interrupt Pending Register" bits described in the FDK user manual. Please refer to the hardware user manual for further information on the possible interrupt bits. The callback function is executed if at least one of the specified interrupt sources occurred.

callbackFunction

This parameter is a function pointer to the user callback function. The callback function pointer is defined as follows:

```
typedef void(*FUNCINTCALLBACK)( TDRV015_HANDLE   hdl,
                                unsigned int      interruptOccurred,
                                void              *param,
                                TDRV015_STATUS    status );
```

hdl

This parameter specifies a device handle which can be used for hardware access or other API functions by the callback function.

interruptOccurred

This parameter is a 32bit value reflecting the occurred interrupts. It is useful if the callback function handles multiple interrupt sources. The interrupt bits correspond to the Infrastructure Module's "Interrupt Pending Register" bits described in the FDK user manual. Please refer to the hardware user manual for further information on the possible interrupt bits.

param

This parameter is the user-specified *funcparam* value (see below) which has been specified on callback registration. This value can be used to pass a pointer to a specific control structure, to supply the callback function with specific information.

status

This parameter hands over interrupt callback status information. The callback function needs to check this parameter. If the specified interrupt source has occurred properly, and no errors were detected, this parameter is TDRV015_OK. If this parameter differs from TDRV015_OK, an internal error has been detected and the callback handling is stopped. The callback function must implement an appropriate error handling.

funcparam

This value specifies a user parameter, which will be handed over to the callback function on execution. This parameter can be used to pass a pointer to a specific control structure used by the callback function.

pCallbackHandle

This value specifies a pointer to a handle, where the callback handle will be returned. This callback handle must be used to unregister a callback function.

EXAMPLE

```
#include “tdrv015api.h”

TDRV015_HANDLE   hdl;
TDRV015_STATUS   result;
unsigned int     interruptMask;
USER_DATA_AREA   userDataArea;
TDRV015_HANDLE   callbackHandle;

/* forward declaration of callback functions */
void callback_TIMER0(   TDRV015_HANDLE     hdl,
                        unsigned int       interruptOccurred,
                        void               *param,
                        TDRV015_STATUS     status);
/*
**  Register callback function for TIMER0 (UINTP0)
**  Use a “normal” priority, and 64KB stack.
*/
interruptMask = (1 << 16);
result = tdrv015InterruptRegisterCallbackThread(hdl,
                     TDRV015_PRIORITY_NORMAL,
                     0x10000,
                     interruptMask,
                     callback_TIMER0,
                     &userDataArea,
                     &callbackHandle);
...
if (result != TDRV015_OK)
{
  /* handle error */
}

/*
** Initialize and start the Timer function, using register accesses.
** Refer to the FDK documentation for register description.
*/
...
```

```
/*
** Callback Function, using API Functions for Register Access
*/
void callback_TIMER0(   TDRV015_HANDLE     hdl,
                        unsigned int       interruptOccurred,
                        void               *param,
                        TDRV015_STATUS     status)
{
  TDRV015_STATUS   result;
  USER_DATA_AREA   *pUsrData = (USER_DATA_AREA*)param;
  unsigned int     u32value;

  if (status != TDRV015_OK)
  {
    /* handle error status */
  }

  printf("[Timer 0 Interrupt]\n);

  /* Acknowledge TIMER0 interrupt source by writing to
  ** "Timer Based Interrupt Status Register" (offset may differ). */
  u32value = (1 << 0);
  result = tdrv015WriteBE32(     hdl,
                TDRV015_RES_MEM_1,
                0x1002C,
                1,
                &u32value );
  /* handle errors */
  return;
}
```

RETURNS

On success, TDRV015_OK is returned. In the case of an error, the appropriate error code is returned by the function.

ERROR CODES

Error Code	Description
TDRV015_ERR_INVALID_HANDLE	The specified TDRV015_HANDLE is invalid.
TDRV015_ERR_INVAL	Function or callback handle pointer is NULL.
TDRV015_ERR_NODEV	Failed to allocate a callback handle
TDRV015_ERR_TASK_CREATE	Creation of the callback thread (task) failed.

3.5.3 tdrv015InterruptUnregisterCallback

NAME

tdrv015InterruptUnregisterCallback – Unregister a User Callback Function

SYNOPSIS

```
TDRV015_STATUS tdrv015InterruptUnregisterCallback
(
    TDRV015_HANDLE      hdl
)
```

DESCRIPTION

This function unregisters a previously registered user callback thread or ISR function.

PARAMETERS

hdl

This value specifies the callback handle retrieved by a call to the corresponding register-function.

EXAMPLE

```
#include “tdrv015api.h”

TDRV015_HANDLE   callbackHdl;
TDRV015_STATUS   result;

/*
** Unregister a callback function
*/
result = tdrv015InterruptUnregisterCallback(callbackHdl);

if (result == TDRV015_OK)
{
   / *OK */
} else {
   /* handle error */
}
```

RETURNS

On success, TDRV015_OK is returned. In the case of an error, the appropriate error code is returned by the function.

ERROR CODES

Error Code	Description
TDRV015_ERR_INVALID_HANDLE	The specified callback handle is invalid.

3.6 Endian Conversion Functions

The following conversion functions can be used to develop endian-neutral software, especially for direct access to mapped PCI resources.

3.6.1 endian_be16

NAME

endian_be16 – big-endian conversion function

SYNOPSIS

unsigned short endian_be16
(
    unsigned short u16value
)

DESCRIPTION

This function converts a short integer value (16-bit) from the native CPU endian order to big-endian order. That means on Intel x86 architectures the value will be byte-swapped, as opposed to PowerPC architectures.

PARAMETERS

u16value

This argument specifies the data to convert

EXAMPLE

```
#include “tdrv015api.h”

unsigned short *pRawData, bigEndianData;

/* setup pRawData pointer to the correct location first */

bigEndianData = endian_be16(*pRawData);
```

RETURN VALUE

This function returns the passed value in the big-endian order.

3.6.2 endian_le16

NAME

endian_le16 – little-endian conversion function

SYNOPSIS

```
unsigned short endian_le16
(
        unsigned short      u16value
)
```

DESCRIPTION

This function converts a short integer value (16-bit) from the native CPU endian order to little-endian order. That means on PowerPC architectures the value will be byte-swapped, as opposed to Intel x86 architectures.

PARAMETERS

u16value

This argument specifies the data to convert

EXAMPLE

```
#include “tdrv015api.h”

unsigned short *pRawData, littleEndianData;

/* setup pRawData pointer to the correct location first */

littleEndianData = endian_le16(*pRawData);
```

RETURN VALUE

This function returns the passed value in the little-endian order.

3.6.3 endian_be32

NAME

endian_be32 – big-endian conversion function

SYNOPSIS

unsigned int endian_be32
(
    unsigned short u32value
)

DESCRIPTION

This function converts an integer value (32-bit) from the native CPU endian order to big-endian order. That means on Intel x86 architectures the value will be byte-swapped, as opposed to PowerPC architectures.

PARAMETERS

u32value

This argument specifies the data to convert

EXAMPLE

```
#include “tdrv015api.h”

unsigned short *pRawData, bigEndianData;

/* setup pRawData pointer to the correct location first */

bigEndianData = endian_be32(*pRawData);
```

RETURN VALUE

This function returns the passed value in the big-endian order.

3.6.4 endian_le32

NAME

endian_le32 – little-endian conversion function

SYNOPSIS

unsigned short endian_le32
(
    unsigned short u32value
)

DESCRIPTION

This function converts an integer value (32-bit) from the native CPU endian order to little-endian order. That means on PowerPC architectures the value will be byte-swapped, as opposed to Intel x86 architectures.

PARAMETERS

u32value

This argument specifies the data to convert

EXAMPLE

```
#include “tdrv015api.h”

unsigned short *pRawData, littleEndianData;

/* setup pRawData pointer to the correct location first */

littleEndianData = endian_le32(*pRawData);
```

RETURN VALUE

This function returns the passed value in the little-endian order.

4 Diagnostic

If the TDRV015 does not work properly it is helpful to get some status information from the driver respective kernel.

The Linux */proc* file system provides information about kernel, resources, driver, devices, and so on. The following screen dumps displays information of a correct running TDRV015 driver (see also the proc man pages).

```
# lspci -v
  …
06:00.0 Bridge: TEWS Technologies GmbH Device 8277
        Subsystem: TEWS Technologies GmbH Device 8277
        Flags: bus master, fast devsel, latency 0, IRQ 17
        Memory at fe400000 (32-bit, non-prefetchable) [size=4M]
        Capabilities: [40] Power Management version 3
        Capabilities: [48] Message Signalled Interrupts: Mask- 64bit+
Queue=0/0 Enable+
        Capabilities: [58] Express Endpoint, MSI 00
        Kernel driver in use: TEWS TECHNOLOGIES - TDRV015 Device Driver
        Kernel modules: tdrv015drv
  …


# cat /proc/devices
Character devices:
  1 mem
  …
250 tdrv015drv
  …


# cat /proc/iomem
00000000-0000ffff : reserved
00010000-0009fbff : System RAM
  …
fe000000-fe7fffff : PCI Bus 0000:02
  fe000000-fe7fffff : PCI Bus 0000:03
    fe000000-fe7fffff : PCI Bus 0000:06
      fe400000-fe7fffff : 0000:06:00.0
        fe400000-fe7fffff : TDRV015
  …
```